SOFTWARE SUPPORT TOOLS

INTERMETRICS

This technical report has been reviewed and is approved for publication.

JOHN C. MOTT-SMITH
Project Officer
Techniques Engineering Division

JOHN T. HOLLAND, Lt Colonel, USAF
Chief, Techniques Engineering Division

FOR THE COMMANDER

STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer
Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-77-126	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): SOFTWARE SUPPORT TOOLS	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER: IR-204-2
7. AUTHOR(s): James L. Felty, Martin S. Rath	8. CONTRACT OR GRANT NUMBER(s): F19628-76-C-0225
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Intermetrics, Inc. 701 Cancord Avenue Cambridge, MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Deputy far Cammand and Management Systems Electranic Systems Divisian Hanscom AFB, MA 01731	12. REPORT DATE: 15 October 1976
	13. NUMBER OF PAGES: 60
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved far Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Support Taols
Software Techniques
High-Order Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is a part of a High-Order Language (HOL) standardization project initiated by the Air Force Systems Command. The purpose af that HOL standardizatian effort is ta determine an Air Farce policy regarding the adaptian of a standard HOL for each of the Air Force saftware applicatian areas, and to develap an implementation plan ta suppart that palicy.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ESD-TR-77-126

SOFTWARE SUPPORT TOOLS

15 October 1976

IR-204-2

James L. Felty
Martin S. Roth

Intermetrics, Inc.
701 Concord Avenue
Cambridge, MA 02138

Prepared For: Contract F19628-76-C-0225
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731

The purpose of this report is the exchange of ideas. It does not represent an official position of ESD or of the Air Force.

This report has been prepared for the Electronic Systems Division, Air Force Systems Command, as CDRL Item A007 under Contract Number F19628-76-C-0225. Mr. John Mott-Smith is the Contract Technical Monitor for the Air Force. Dr. Larry Weissman is the Program Manager for Intermetrics, Inc.

The authors wish to acknowledge the valuable information provided by numerous persons at the Rome Air Development Center, the Air Force Avionics Laboratory, the Space and Missile Systems Organization, the MITRE Corporation, and the Aerospace Corporation.

Special acknowledgements are given to Dr. James S. Miller of Intermetrics for his technical assistance and to Mrs. Deborah Gleason and Miss Judith Haigh for typing the manuscript.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Scope and Description of This Report

This report is part of a High-Order Language (HOL) standardization project initiated by the Air Force Systems Command. The purpose of that HOL standardization effort is to determine an Air Force policy regarding the adoption of a standard HOL for each of the Air Force software application areas, and to develop an implementation plan to support that policy.

In order to arrive at the most appropriate policies and plans, the Air Force HOL standarization program includes an examination of the entire software development environment in order to assess various potential impacts on standardization and language selection. The availability and use of software engineeering techniques and tools are an important part of that environment. The availability of tools for a given language is a factor that should be taken into account when considering a language as a potential standard. Likewise, a decision to standardize on a particular language will affect what tools are necessary.

This report distinguishes between software "techniques" and software "tools": a "technique" is any method employed by software engineers, designers, or programmers in the creation, use, or maintenance of software. These include managerial techniques such as "chief programmer teams", software engineering techniques such as "top-down" design, and "structured" programming. A software "tool", on the other hand, is itself a piece of software -- a computer program that aids in the preparation of other programs. These include compilers, simulators, and code analysis and diagnostic aids of all sorts.

Software tools are the subject of this report. Software techniques are the subject of a parallel report prepared by the MITRE Corporation [Cheng 1976]. Later in this section, an overview of the current tools situation will be given in order to provide the context for the remaining sections of the report.

Section Two consists of a taxonomy of major tool categories, with examples of significant kinds of tools within each category. For each kind of tool, a brief definition is included.

In Section Three, the taxonomy is illustrated with specific examples of tools. Most of these example-tools are owned by the Air Force and for the most part they are directed toward applications programmed in JOVIAL. The information on each tool is condensed to a few paragraphs and describes a) the tool's purposes, b) its value, and c) its degree of portability.

Section Four discusses several tool-related topics, drawing examples from the previous sections. These topics are: a) the portions of the Computer Program Life Cycle given in [AFR 800-14 1975] that are best and least supported by tools, b) some ways in which tools and HOL standardization affect each other, including the effect of standardization on the need for certain tools, and c) the trend toward the design of integrated tool "packages".

This report does not attempt a complete listing of all the kinds of tools ever used, nor does it purport to be a complete catalogue of all Air Force tools or tools packages. No such list exists, and the task of preparing one is beyond the scope of this report.

1.2 Overview of the Current Tools Situation

Tools are the most widely used computer programs [Fisher 1976]. It is estimated [Callender et. al. 1975] that twenty percent of Air Force software costs are for software support tools. According to [La Padula and Loring 1976], the existence of a particularly important tool -- a compiler -- has been a factor in the selection of what language to use for various Air Force projects. For most application areas, the existence and availability of a compiler for a language has been a strong argument for the use of that particular language.

The application area of operational flight programs (including avionics) presents a particular need for tools. The consequences of program failure in these applications is more expensive or catastrophic than they are in some other application areas. Also, the efficiency of the compiled code is highly important. Here, the existence of certain simulation and verification tools on larger host computers, in addition to compilers, has influenced the choice of language [La Padula and Loring, 1976].

Software contractors frequently use whatever tools they have available (sometimes proprietary ones), or they develop ad hoc tools that may not be re-applicable or re-usable. These are often re-invented or re-built for each new project. Unfortunately, comprehensive information on what tools are being used (and have been used) by all Air Force contractors does not exist. According to [Fisher 1976], a vendor may write software tools for application software in his own proprietary language, strengthening the ties of the vendor with the application software he has developed. Furthermore, the major purpose of a given software project is to develop the application software, and scarcity of funding or lack of time often prevents the development of more generally applicable tools.

Tools such as compilers, testing systems, and various diagnostic aids are frequently language-dependent, and each different language requires that a new set of tools be developed. For this reason, the Institute for Defense Analyses [Fisher 1976] believes that one of the immediate savings offered by standardizing on a common high-order language will be felt in the area of software tools, including compilers.

The Air Force is now developing its own tools, primarily but not exclusively tools which process the versions of JOVIAL and of other languages which are emerging as common Air Force languages. The Rome Air Development Center has been involved in tools development for several years, and the tools that RADC has initiated are now becoming available and are being publicized [RADC 1976b]. Many of the tools described in Section Three, below, are RADC products. However, many of the most significant Air Force-developed tools have yet to be used on operational software, either by contractors or by the Air Force.

No well-stocked "library" of tools exists, although tool libraries have been started in the past (see for example, the "Aids Inventory" project reported in [Reifer 1975b]). Several tools are, however, available on the ARPANET [Feinler 1975]. The National Software Works exists as a potential repository for the distribution and meantenance of tools, but at present it contains less than a dozen tools, including compilers [Carlson 1976]. Language commonality and tool re-usability may help such repositories more fully realize their potential usefulness.

2.0 SOFTWARE TOOLS TAXONOMY

This section categorizes the more important kinds of software tools and provides a brief definition of each. They are grouped according to their function in such a way as to minimize the overlap between categories. The taxonomy does not purport to be a complete listing of every kind of tool ever devised. Other glossaries and listings of tools are given in [Reifer 1975a and 1975b].

The structure of the taxonomy is that of the following top-level hierarchy:

I. Translators and Aids Related to Development and Maintenance of Languages and Translators

   A. Translators
   B. Translator development and maintenance aids
   C. Language development and maintenance aids

II. Application Program Aids

   A. Aids in application program design
   B. Aids in application program development, test, and maintenance

      1. Aids in program preparation
      2. Program environment aids
      3. Program code analyzers

         a. Static code analyzers
         b. Dynamic code analyzers

      4. Aids for combining programs into systems

Software support tools are categorized in terms of the above structure as follows:

I. Translators and Aids Related to Development and Maintenance of Languages and Translators

A. Translators

1. *Compilers* -- programs which translate a program written in a high-order language (a "source" program) into an equivalent machine-language program for subsequent linking, loading, and execution. A compiler contains a syntax analyzer (the compiler's "front end"), which parses source program text, and a code generator (the compiler's "back end"), which emits machine-language instructions for the computer on which the program is to be run. A compiler is the basic tool for a production language. If a compiler is not available for a given machine, the source language is not available on that machine. Furthermore, without a compiler, no other tools will be available on that machine.

2. *Optimizers* -- routines or groups of routines that modify a program so that it will occupy less storage, or will execute more efficiently, without altering the program's meaning. Typically, the optimizer is part of the compiler.

3. *Preprocessors* -- programs that take as input a source program and produce as output a modified version of the source program that is suitable as input to a compiler. Typically, a preprocessor allows the programmer to use structured language constructs which are translated by the preprocessor into equivalent unstructured forms. Preprocessors of this type are plentiful and varied, particularly for FORTRAN, a language that was designed prior to the development of more modern, "structured" languages. (A list of over fifty available FORTRAN preprocessors is given in [Reifer 1976].)

4. *Loaders* -- programs that resolve external references and assign absolute addresses to relocatable code, so that the resulting code may be loaded into memory. Loaders accept as input the output of one or more compilations and produce as output an operational program. Loaders are readily available. Some kind of loader is required in order to use a language on a machine. Their sophistication varies, though, according to the machine and environment.

5. *Cross compilers* -- compilers that run on a host machine and generate code for another machine, the "target" machine. The use of cross-compilers is common in military applications which use small target computers that do not support compilers. Cross-compilers may play an important role in language standardization in that they facilitate making a common language available on a variety of machines.

B. <u>Translator Development and Maintenance Aids</u>

1. *Compiler validators* -- programs or sets of programs which, when processed by a compiler, produce results that help demonstrate whether the compiler correctly implements the language for which it was written. Compiler validators exist for FORTRAN, COBOL, JOVIAL (J3), and JOVIAL (J73/I). (see below, pp. 19 and 39). A successful language standardization effort must include some form of compiler validation. A new common language would require that a new compiler validator be developed.

2. *Compiler-writing tools* -- programs that accept, as input, the specifications of a language and produce, as output, portions of a compiler which implements that language. These methods are comparatively new, having appeared within the last decade. Automated construction of syntax analyzers is now well understood, and several systems are widely available and commonly used (e.g., XPL, AED). The trend is toward including more capabilities, such as optimization and error recovery, into the tools. JOCIT and CWS, described in Section 3.0, and examples. All compiler-writing tools, however, still require that the code generators be written by hand.

C. <u>Language Development and Maintenance Aids</u>

1. *Language specification languages* -- languages designed for the writing of formal specifications of the syntax and/or semantics of a programming language. For the most part, language specification languages are still in the developmental stage. The most successfully used language specification language is the Vienna Definition Language (VDL), which is not a tool per se since it is not processed by machine. (The ANSI standard for the PL/I language is written in an English version of VDL.) The SEMANOL system (described in Section

3.0, below), contains a language specification language that can be processed by machine, but SEMANOL, too, is still experimental.

2. *Language specification processors* -- processors that accept a source program and a formal specification of the language in which that program is written, and produce the results of running that source program. The only language specification processor developed so far is the SEMANOL processor. SEMANOL will not be used except experimentally, at least in the near future.

II. Application Program Aids

A. Aids in Application Program Design

1. *Program design languages* -- languages in which programs may be written that formally represent the initial or high-level design of a source program or system. Upon completion of the design, programs written in program design languages can be expanded by the programmer into the appropriate lower-level source code. Program design languages as a separately identifiable tool are seldom used, although programmers often write down a top-level design using a mixture of English and programming language constructs.

2. *Simulation modeling aids* -- tools that aid in the construction of models that simulate the running of a completed system of programs. They aid in determining probable system performance and help evaluate alternative system design. Modeling is typically used prior to hardware purchase to ascertain the most cost-effective acquisitions. Less frequently, it is used to determine constraints on separate program modules in time-critical applications. (See, for example, [Ramamoorthy 1965] for Markov modeling of programs.) There exist languages that are specifically geared to simulation modeling (e.g., GPSS). Additionally, subroutine packages are available for this purpose.

B. Aids in Application Program Development, Test, and Maintenance

1. Aids in Program Preparation

a. *Text editors* -- tools that allow a programmer to build a file containing a computer source program. They allow the user to add, delete, or change lines or portions of lines of a program. Often, they operate interactively. Some text editors are language dependent and perform some correctness-checking and allow insertion or deletion on the basis of syntactic units. Text editors are widely available and form an integral component of most software development systems. They are most frequently coded in assembly language and are machine specific.

b. *Program support libraries* -- sets of routines or programs which perform various storage and maintenance functions facilitating the organization and control of programming data. Program support libraries are widely used in industry. A good example of a language that requires this type of software tool is COBOL. The indexed access features of this language necessitate a set of routines for periodic reorganization of files stored in this format. A program support library is described in Section 3.0.

c. *Programming macros* -- commonly-used (sometimes parameterized) sequences of instructions that a programmer may cause to be inserted into his program at appropriate points. These may be provided in library form to assure the consistency of data representation between different modules and programs by making a single definition commonly available (e.g., the COPY mechanism in COBOL).

d. *Source program reformatters* -- programs or routines that accept, as input, programs written in any format legal for that language and produce as output that same program reformatted according to a consistent, predetermined system of spacing and indentation. The reformatted output is often called "pretty printing". Both the PASCAL and SIMULA 67 languages provide this facility on the DEC system-10.

2. Program Environment Aids

a. *Dynamic environment simulators* -- programs that test an application program by simulating the application program's actual environment. These respond to the application program's outputs by supplying the program with inputs which reflect the environment's response to those outputs. These allow the user to determine how the program will behave in various kinds of real situations. Some form of dynamic environment simulator is necessary in order to test avionics programs before the software is actually airborne. The SDVS system described in Section 3.0 contains such a simulator.

b. *Stub generators* -- programs that facilitate top-down programming by providing "stubs" -- i.e., dummy routines that may be called by the higher-level program in order to test the higher-level program prior to the actual coding of the lower-level routines. Some stub generators simulate the storage or execution-time requirements of actual routines. This constitutes one feature of IBM's Program Support Library (PSL) described in Section 3.0.

c. *Drivers* -- programs that simulate a higher-level program by invoking and testing the actual coding of lower-level routines. Whereas stub generators facilitate "top-down" programming, drivers facilitate "bottom-up" programming. There are no generators for drivers; each driver must be specially contrived for the application.

d. *Test case generators* -- programs that help the user discover a set of input data which exercises a specified path in the program. These tools are still experimental, and they cannot be used to verify that all paths have been tested. Sometimes, the term test case "generator" is used as a synonym for "driver." (See, e.g., [Pomeroy 1972].)

e. *Statement-level simulators* -- programs that process compiled source code on a host computer in such a way that for each source-level statement, the execution characteristics approximately equal those of the target computer for which the source code is ultimately intended. The SDVS system described in Section 3.0 contains such a simulator. Statement-level simulators facilitate software design

without a target computer. They reside on large machines which permit use of additional aids that the smaller target computers do not support. In some cases, statement-level simulators facilitate design and test of very complicated and high performance algorithms which may eventually be coded in assembly language. The HOL implementation can serve as a design aid for, and provide test case results to, the assembly language version. Also, the use of an HOL allows the use of a statement-level simulator rather than an interpretive computer simulator (see below), which operates a couple of orders of magnitude more slowly.

f. *Interpretive computer simulators* -- programs that enable code for a target machine to be exercised on a host machine with the register-level, bit-level, and memory-cycle instruction timing accuracy of the target machine. Whereas a statement-level simulator executes one source statement at a time, an interpretive computer simulator executes one machine instruction at a time. If the actual target machine hardware is not available during the software development process, this kind of simulator is essential. The SDVS system described in Section 3.0 contains an interpretive computer simulator.

3. Program Code Analyzers

a. Static code analyzers

(1) *Cross-referencers* -- programs that identify, for each symbol or name appearing in a source program, all of the source-level statements in which that symbol or name appears. Often, read-references are distinguished from assignments. Quite frequently, declarations are also specifically flagged. Cross-referencers are extremely common and valuable tools for maintenance of large programs.

(2) *Static statistics gatherers* -- programs that compute statistics based on the number of times various items appear in a source program. These are growing in use. See, for example, the JOVIAL (J3) Statistics Collector description in Section 3.0.

(3) *Flow charters* -- programs that accept as input a source program and produce as output a flow chart of that source program. They help one determine the structure of existing programs and are available for languages such as FORTRAN that lack modern structuring facilities.

(4) *Code auditors* -- programs that examine source code and notify the user as to whether or not that source code was written in accordance with certain pre-determined programming standards or conventions. These are valuable in large multi-programming projects to ensure uniform and readable coding. An example of this tool is the FORTRAN Code Auditor described in Section 3.0.

(5) *Programming standard enforcers* -- programs or routines that examine source code and prevent the programmer from using certain language constructs or programming techniques that the language itself does not preclude. Programming standards enforcers may be used in conjunction with a compiler for the full language in order to achieve the effect of a subset implementation. The most common use of this type of tool is in conjunction with large and complex languages (e.g., PL/I).

(6) *Overlay analyzers* -- programs that examine source programs in order to determine mutually disjoint segments that can reside in the same area of memory at run time.

(7) *Automated program provers* -- programs that assure the correctness of a source program through proof of a set of assertions on all outputs. This proof utilizes axioms of computer arithmetic, assertions about the input variables, and usually intermediate assertions pertaining to the values of variables at specific program points. The assertions are specified in predicate calculus, and proofs are generated by automated theorem provers. The appeal of this approach rests in the fact that correctness is <u>proved</u> for <u>all</u> possible inputs. Due to the limited power of presently

available theorem provers and the unlikelihood of any major breakthroughs in this area, commercial program provers will not emerge in the next ten years. The current state of these and related tools is discussed in [King 1976].

b. Dynamic code analyzers

(1) *Program debuggers* -- programs or routines that provide diagnostic information useful in locating errors in a source program. Debuggers commonly provide the following kinds of information: dumps, which record the complete state of execution (memory and register contents) at some point, usually the point of termination; snaps, which record intermediate values of certain items during execution; traces, which record the state transitions that occur during execution; and breakpoints, which interrupt normal computation and cause debugging activities to commence. Debuggers may operate either in batch mode or interactively. Program debuggers are widely used and exist for almost every high-order language. Some debuggers operate at the source level (e.g., SIMDDT for SIMULA 67 and COBDDT for COBOL on the PDP-10) and others at the object level with some additional source information (e.g., DDT for FORTRAN on the PDP-10).

(2) *Program path execution analyzers* -- programs that augment source code by adding counters and various other statistics-gathering indicators, then execute the augmented code and produce reports on how thoroughly the various portions of the source code have been exercised. These tools thereby aid in determining when source programs have been tested adequately, and what kind of test cases need to be submitted in subsequent runs. In addition, this information can be used for determining frequently-exercised program modules, so as to isolate the most valuable sections to optimize. Many path analyzers have been developed. A comparative analysis of sixteen such tools (available commercially and from universities) has been published [Kennedy 1974].

(3) *Symbolic executors* -- programs that execute source code, not with real input values, but with "symbolic" values -- mathematical expressions of what the range of possible values is. The effect is that of exercising the program for each of the various possible data values. The result of symbolic execution of a statement is a symbolic expression which is a function of the program's (symbolic) inputs. Symbolic executors are still in the class of research projects. Their future application will most probably be in the areas of debugging and optimization. EFFIGY, an experimental symbolic executor, is described below in Section 3.0.

4. Aids for Combining Programs into Systems

a. *Implementation monitors* -- programs that examine the code of the various modules of an application system of programs and maintain a data base from which reports are generated concerning system status and subsystem interactions. This class of tools finds its greatest use in large software projects, for which they can greatly simplify the management problems involved. SIMON, described in Section 3.0, is such a tool. Additionally, CARA and PSL (also described in Section 3.0) contain implementation-monitoring capabilities.

b. *Interface checkers* -- programs that examine data utilization across separately-compiled modules and determine whether the accessing module interprets the accessed information in the intended fashion. As the goals of program modularity and reliability are more widely achieved, interface checkers will see greater use.

c. *Recompilation-necessary identifiers* -- programs that, given a change in a compool module, determine which modules having potential access to that data in that compool need to be recompiled.

d. *Linkage directories* -- programs that are used in the absence of dynamic linkage to determine which modules of a program system need to be loaded together. This type of tool is extremely valuable when a programmer is forced to specify explicitly all required modules at load time and when many interconnected modules are in use (possibly written by different programmers). Such a tool exists for SIMULA 67 on the PDP-10.

3.0 EXAMPLES OF SPECIFIC SOFTWARE TOOLS

This section describes some specific tools in terms of the taxonomy presented in Section Two. Most of these tools are under Air Force ownership and control. Taken together, they represent a significant portion of the Air Force's involvement in a comprehensive tools program, and they indicate the force and direction of current software tool development.

The information presented on each tool is limited to the following topics:

1) The purpose and function of the tool;
2) The estimated value of the tool; and
3) How the language, project, or machine dependencies of the tool affect the tool's re-usability.

Because the discussions are brief, references to additional documentation are provided where available.

JOCIT

(JOVIAL Compiler Implementation Tool)

Developed by Computer Sciences Corporation and owned by the Air Force

JOCIT is a compiler writing system for JOVIAL (J3). It automatically generates the entire compiler front-end. This includes the parser, the scanner, error recovery, and an extremely powerful code optimizer and constitutes about 70% of the compiler. These sections when combined translate a JOVIAL (J3) program into Intermediate Language (IL). A separately written code generator, then, accepts the IL instructions and produces machine code.

The tool is specifically designed for version J3 of JOVIAL. The decision to write JOCIT stemmed partly from the inappropriateness of JOVIAL (J3) for compiler writing. Other JOVIAL dialects are being implemented in themselves (e.g., J73/I at Wright-Patterson) and therefore extensions of JOCIT for other versions of JOVIAL currently are not being considered.

JOCIT is written primarily (95%) in SYMPL. The remaining 5%, containing all machine-dependencies, is coded in assembly language. The tool was originally developed on a UNIVAC 1108 and is presently operational on a HIS 6180 running under GCOS. The prototype version available on the 1108 is only slightly different from that in use on the 6180. JOCIT requires approximately 45K 36-bit words of storage, but could be reconfigured for a smaller machine as 96K 16-bit words. Rehosting of JOCIT is considered a minimum of a one man-year project.

The original version of JOCIT was used by CSC to produce J3 compilers for the CDC 6400 and IBM 360. RADC, who was responsible for all Air Force JOVIAL (J3) compilers, has used JOCIT to build a compiler for the Honeywell 6180. A prototype version of JOCIT was available at RADC as early as December 1972. The final version was delivered in August 1974. From that time until about one year ago, when CTCC took over maintenance, RADC was using the tool to generate a new release approximately every three months. The ease of production of these new releases is one major virtue of JOCIT.

The true value of JOCIT, however, rests in its use to produce new compilers. It is estimated that without JOCIT, production of a JOVIAL (J3) compiler would require two years and between $500,000 and $1,000,000.

The higher estimate reflects the cost of a compiler incorporating all optimizations which are automatically supplied by JOCIT. In contrast, making use of JOCIT, a new compiler can be constructed in between six and nine months at a cost of about $200,000 [DiNitto and Motto 1976]. In addition, due to the already thorough testing of JOCIT, its compiler would be more reliable.

Further information on JOCIT may be found in [CSC 1975], [CSC 1976], and [Dunbar 1975].

CWS
(Compiler Writing System)
Developed by Aerospace Corporation and Owned by the Air Force

CWS is a compiler writing tool, derived from SPLIT (Space Programming Language Implementation Tool). SPLIT was originally designed by Systems Development Corporation solely for implementation of SPL. After proving its worth in this context, Aerospace Corporation was granted a 2-year contract to improve and enhance the basic system for more general use. CWS is particularly geared to compiler construction for languages used in real-time, avionics applications.

Two languages are used for constructing compilers in conjunction with CWS. SAL, syntax analysis language, is used for writing the front-end or parsing section of the compiler. SAL provides facilities specifically applicable to this area (e.g., dictionary construction and maintenance routines). The second language, GN, is designed for implementing the code generation section. A standard intermediate language is utilized for communication between the front and back-ends of the compiler. This permits several parsers to use a single code generator or a single parser to be linked to various code generators for different machines [Lesak 1976].

The tool runs on both the CDC 6600 and CDC 7600. It has been used in the implementation of several languages including SPL, HAL/S and OPAL. It is estimated that the initial cost savings to the government from CWS's use has been in excess of $750,000. The implementation of a syntax analyzer for a non-ambiguous language specified in Backus-Naur form can be achieved in one man-month. Using CWS, two persons can write a code generator in approximately six months.

CWS is available from SAMSO at no charge. Manuals on the system include an internal maintenance guide, several user's manuals and a tutorial in its use. The system is described in [Corn 1974].

JCVS
(JOVIAL Compiler Validation System)
Developed by ABACUS Corp. for RADC

JCVS consists of six modules of JOVIAL source program statements which test whether a compiler correctly implements the JOVIAL language. Two versions of JCVS exist: one for J3 (the AFM 100-24 version) and one for J73/I (the version to become MIL STD 1589). The first of the six test modules contain tests of the language features. The predominant number of tests is in this category. The tests are organized in terms of the paragraphs in the language reference manual, to aid in identifying what features failed. Each test is annotated with source program comments, and the printed output analyzes the results and specifies possible corrective action. The other modules test for error detection, capacity, efficiency, and machine and implementation features.

The (J3) JCVS effort began at ESD about fifteen years ago, but its major development was in 1968-69. It has been in active use since 1971. Its first serious use in validating compilers so they could pass tests was by WWMCCS in 1972. It has now been applied to most existing J3 compilers, including the JOCIT compiler on the HIS 6000 under GCOS. RADC maintains the J3 version. Revisions to JCVS are still being made.

The J73/I version of JCVS has been used on the DEC-10 version of the J73/I compiler. DAIS uses and maintains the J73/I version [DiNitto 1976].

Portability of JCVS is enhanced by the fact that most machine dependencies are localized to preliminary DEFINE declarations which precede the data declarations and the test modules.

Modifying JCVS to test a compiler for another language would require re-coding JCVS again from scratch. (Even the J73/I version of JCVS entailed a recoding rather than a revision of the J3 version.) One rough estimate obtained was that it would require a 2-3 manyear effort to produce a non-JOVIAL compiler validation system.

STRUCTRAN-1 and STRUCTRAN-2 Programming Translators
Developed by General Research Corp. for RADC & DMA

These translators are tools which facilitate structured programming in FORTRAN. STRUCTRAN-1 accepts structured FORTRAN programs written in DMATRAN and translates them into ASA standard FORTRAN X3.9. In addition, the source code is indented to reflect its structure, so as to increase readability. DMATRAN supports five control structures: if then else, do while, do until, case, and invoke. The invoke statement permits use of internal subroutines, thereby enabling greater modularization.

STRUCTRAN-2 accepts Univac 1108 Fortran IV programs and translates them into logically equivalent DMATRAN programs. The tool is unique in this respect [Mark 1976]. It permits existing programs to be converted so that an entire project may be carried out within a structured environment. Both translators are written in FORTRAN and were originally developed on a Honeywell Series 600/6000 computer running under GCOS. They are presently run on a Univac 1108 at the Defense Mapping Agency. At this installation, they have been available since March 1976 and receive daily use. STRUCTRAN-1 requires approximately 50K words to run on the HIS 600/6000. STRUCTRAN-2 is chained (overlaid) and runs in about 72K words.

Transport of STRUCTRAN-1 presents no real problems, since it is written in FORTRAN. STRUCTRAN-2 will be somewhat more difficult to transport.

Users and maintenance manuals for both will be published shortly and will be available from RADC. Both tools are maintained at RADC/ISIS by Donald Mark.

SEMANOL
(SEMANtically-Oriented Language)
Developed by TRW Systems, Inc., for RADC

SEMANOL offers an interpreter-oriented approach for formally defining programming languages. It consists of the SEMANOL language specification language and a language specification processor. The language specification processor accepts as input a SEMANOL program specifying a programming language, a source program written in that specified language, and the data for that source program. The output is the output of the source program, executed according to the SEMANOL language specification for that language.

SEMANOL has been used to define JOVIAL (J3) [TRW 1973a] and a partially completed SEMANOL definition of J73 exists [TRW 1975]. In its current form, the practicality of the SEMANOL processor is limited in that execution is very slow and costly: both the source program and the SEMANOL definition of the source language are processed interpretively. For each execution of a source program statement, the SEMANOL language specification of the source language must be re-processed. In addition, the coding of the source language definition in SEMANOL is a difficult, lengthy manual process. For these reasons, plus funding problems, SEMANOL receives little use at present. Nevertheless, this kind of tool is potentially of considerable value to implementers, controllers, and maintainers of a programming language.

The current SEMANOL system is coded in FORTRAN and runs on a Honeywell 6180 machine under MULTICS. In theory, SEMANOL may be used without modification to define and process other languages, since it operates on definitions of languages. Any definition processed by the SEMANOL processor must, though, be manually coded in SEMANOL [TRW 1973b].

CARA

(Computer Assisted Requirements Analysis)

Developed by the University of Michigan, modified and owned by ESD

CARA is a two-part system for defining and reporting on the hardware and software requirements of proposed projects. It was originally developed by the University of Michigan under the title of PSL/PSA (problem specification language - problem analysis language). ESD purchased PSL/PSA and modified it. The two parts of the system are presently entitled User Requirement Language (URL) and User Requirements Analysis (URA). CARA can be used by the system program office during the acquisition cycle to flesh out the system specifications as supplied by headquarters. It would also be used after product delivery to evaluate whether or not all requirements have been met.

CARA is written in FORTRAN, PL/I, and assembly language. It runs on a Honeywell HIS 6180 under MULTICS. The transport of the tool to any other machine is seen as difficult [Melanson 1976]. ESD will supply the system to any interested sites but will not assist in its installation. CARA also runs on the IBM 360 TSO.

A prototype of the system has been in use for two years. It is presently installed at RADC and SAMSO. The Navy also has used the tool. A preliminary evaluation has been written. Several additional evaluations have since been written by MITRE. Development of CARA is continuing to facilitate specification of problem areas of particular interest to DoD (e.g., security). A similar system is being designed by TRW, but concentrates primarily on the area of ballistic missile construction.

Two manuals are available from ESD detailing URL and URA. They are entitled URL User's Manual [ESD 1975a] and URA User's Manual [ESD 1975b].

PSL
(Program Support Library for WWMCCS)
Developed by IBM for RADC

PSL is a program support library designed to aid top-down implementations. Phase I (to be delivered in November, 1976) contains the following capabilities: (1) a COBOL preprocessor, (2) management report production facilities (both internal and external listings), (3) an automatic stub generator, (4) a source program reformatter that provides automatic indentation, (5) an INCLUDE capability, for unit programming, and (6) various library maintenance features, allowing one to add and delete units and to execute various sections of the library.

Five WWMCCS sites have been selected for testing Phase I of the PSL. Within a year, all WWMCCS sites should be using it [Ruple 1976]. AFSC Hdq. (Andrews AFB), the Defense Mapping Agency (DMA), and several SAMTEC SPOs have plans to use PSL in the future. No formal evaluation of PSL has been written, but an evaluation based on the test sites' usage will be prepared.

A "Phase II" version of PSL has been planned and tentatively scheduled for April 1977, but it as yet has no sponsor. Phase II PSL would include FORTRAN and JOCIT JOVIAL capabilities.

PSL is coded mostly (approx. 95%) in COBOL, with a few GMAP (assembler) routines. It is designed to run on the Honeywell 6000 series machines under the WWMCCS 6263 operating system. Phase I contains 22K lines of (COBOL) code, requiring approximately 44K words on the 6000. It will require modification before being transported to a different machine and operating system. It was roughly estimated that converting PSL for DMA's use on the Univac 1108 would cost on the order of $60-70K. To modify PSL for additional language capabilities would be easier. It was roughly estimated that an existing FORTRAN preprocessor could be integrated into PSL for approximately $2K, and that to modify PSL to support an additional language (e.g., SIMSCRIPT) would require on the order of $10-15K. The entire Phase I PSL development contract was approximately $450K.

The functional requirements and the program specifications for PSL are published in [Luppino and Smith 1974] and [Luppino and Tinanoff 1974], respectively. A Phase I User's Manual has been published [IBM 1976].

SDVS
(Software Design and Verification System)
Developed by TRW Systems, Inc. for AFAL

SDVS is an integrated set of tools designed to aid in the development, testing, and maintenance of avionic mission software. It contains a variety of simulation and management aids, including: executive control, a statement-level simulator, an interpretive computer simulator, a multiplex data bus simulator, a linker/loader, and facilities for file generation, post-run analysis (including snapshot and rollback), and flight scenario generation. A high-level simulation language is provided for the creation of flight scenarios, which can be checked through all phases of the system. Facilities exist for run-time control and for post-run data reduction and analysis [TRW 1976].

SDVS was created primarily for use in the development of DAIS avionics mission software on the DEC-10 at AFAL. It is coded in J73/I and processes flight software written in J73/I. (Some SDVS support is, however, coded in assembly language and some in COBOL.)

The final version of SDVS ("Phase 3") under the development contract was scheduled for completion and delivery on June 30, 1976. Under the current maintenance and enhancement, some modifications are being made that will make SDVS easier to use. Notably, the file-handling and cataloguing capabilities will be made more flexible and less cumbersome.

SDVS has yet to reach its anticipated level of productivity on DAIS, though, for two main reasons: (1) DAIS software development has proceeded in parallel with the development of SDVS, and by the time phase 3 SDVS was delivered, the mission software was well along in development using other means of testing. (2) SDVS as it exists lacks the versatility and flexibility necessary in developmental phases of the software process. The SDVS developer and potential DAIS user are (broadly speaking) in agreement on what needs to be done to make SDVS easier to use, and several of the modifications are planned for the current SDVS maintenance contract [Price 1976, Trainor 1976].

Other potential SDVS users include the AF Flight Dynamics Lab and the Eglin AFB (Florida) weapons management program.

Although SDVS was designed to be "highly transportable" [Hollowich and Borasz 1976], re-hosting has not been attempted. A TRW contact roughly estimated that rehosting SDVS on the IBM 360 or the Univac 1108 would be something on the order of a 150-200 man-month effort. An AFAL contact roughly placed the figure at more than $100K and less than $500K, assuming that the new system had a text editor, a DBMS, and a J73/I compiler. (The SDVS development contract itself was bout $1 million.)

As for modifying SDVS to process software written in a language other than J73/I, the TRW person's estimate was roughly 12-14 man-months, and the AFAL person's estimate was at least $100-150K. To re-code SDVS itself in another language (e.g., FORTRAN or PASCAL) would require on the order of 60-70 man-months (TRW estimate) or $200-300K (AFAL estimate).

Changing the target computer would entail changing the interpretive computer simulator. The TRW contract estimated that that would require 8-10 man-months; the AFAL person's estimate was $15-20K.

All of the above figures were off-the-cuff guesses. They indicate roughly the size of the job; they are not definitive, nor do they represent TRW or Air Force commitments.

PALEFAC

Developed by C. S. Draper Laboratory for AFAL

PALEFAC is a tool that sets up the tables needed by the operating system of on-board avionics computers. PALEFAC is run before the plane takes off, not in real time. Input to PALEFAC consists of (a) the source code of the mission software, (b) knowledge of which modules go in which processors, (c) filenames of modules, (d) a list of all bus messages, and (e) general global information about the system. Output of PALEFAC consists of (a) executive tables in source code form and (b) a list of modules to be linked, including the PALEFAC-produced executive tables (in a form suitable for input to a linker). PALEFAC assumes that there can be only four processors maximum in the computer system. PALEFAC is unique in that it does not set up the executive tables in real time [Chalstrom 1976].

PALEFAC is being developed for use by the DAIS executive. PALEFAC is not completed yet, so it is too early to assess the tool's value in an operational setting.

PALEFAC is being written in J73/I to process J73/I on the DEC-10 under TOPS. It could be transported to any system having a J73/I compiler, but it would be very difficult to modify PALEFAC so that it would process another language [Chalstrom 1976].

Functional design specification [Draper 1976a] and detailed design specifications [Draper 1976b] have bcen prepared.

JOVIAL (J3) Statistics Collector
Developed and owned by RADC

The JOVIAL (J3) Statistics Collector is a tool for measuring the frequency with which various features and constructs of JOVIAL (J3) are used. Counts, averages, and percentages are obtained for given source programs regarding the types of data, size, and nature of arrays and tables, and type of statements used. In addition, information on the use and nesting of complex statements, amount of commenting, and compool references is made available.

The tool is meant to help the programmer, project manager, and future compiler designers. The programmer can be discouraged from use of inefficient constructs. Modules containing insufficient commenting can be quickly identified by the project manager. The ease of determining the use of all compool references is also seen as helpful to the project manager [Stover 1976]. By pinpointing the frequently used features of the language it is felt that the tool will facilitate better implementations of the language.

A prototype system is presently under development at RADC on a Honeywell 6180 computer running under GCOS. The system will require approximately 50K words of core storage. It is being written in JOVIAL (J3) and will be transportable without great difficulty to any machine which supports this language [Stover 1976]. Its use with other languages is not foreseen. Other statistics collectors are being developed for this purpose (e.g., BASIC Statistics Collector).

An interim report on the JOVIAL (J3) Statistics Collector is available from RADC/ISIS. CSC describes the characteristics they feel are needed in a JOVIAL statistics collector in [CSC 1976].

BASIC Statistics Collector
Developed and owned by RADC

This tool is used to collect statistics about language feature utilization. It maintains a master file on all programs which have been run through it containing the number of occurrences of statements of each type. The input is a BASIC program. As new unsupported versions of BASIC are encountered, the system is expanded to process them.

The BASIC Statistic Collector is written in PL/I and runs on a Honeywell 6180 computer under MULTICS. The memory requirements are small; less than 11K words. Transport beyond the MULTICS environment is seen as impossible [White 1976b]. Its ability to support other languages is limited to new versions of BASIC.

The tool has been in use at RADC for two years and several other installations have sent their programs there for processing. The results of the project (collected statistics) were forwarded to Dick Nelson who presented them to the ANSI BASIC Committee. These results are presented along with a description of the tool in [White 1976a]. The statistics collector was developed primarily for the production of that report and has not been used since (i.e., in the last six months).

EL and PL - Two Languages for Estimating Program Efficiency
Developed by Jacques Cohen and Carl Zuckerman of Brandeis University

These two languages constitute a system for estimating the run-time of ALGOL programs. The PL-processor translates a given program into a time-formula. This formula represents the time required for execution in terms of HOL operations, branch probabilities, and loop counts. The HOL operations are represented by symbolic time-variables (e.g., if-overhead, integer-addition, assignment). EL is a symbolic manipulation language which permits the user to bind values to the different variables in the time-formula, and to print and plot the results.

The system runs on a PDP-10 under TOPS-10. The PL-processor is written in FORTRAN and comprises approximately 1,000 lines of code. Transport to any system supporting FORTRAN is straightforward. This translator requires 23K words of storage on the PDP-10. EL consists of approximately 1,200 lines of ALGOL-60 code and runs in 35K. The portability of this program is decreased by use of non-standard string and input-output operations, but conversion should require less than one man-month. In addition, compiler-dependent optimizations and values for machine-dependent time-variables would have to be included.

The tool is available, at no charge, from Jacques Cohen of Brandeis University. Its description appeared in [Cohen and Zuckerman 1974].

FORTRAN Code Auditor
Developed by TRW Systems, Inc., for RADC

RADC's FORTRAN code auditor, when provided with a FORTRAN source program, automatically enforces pre-established FORTRAN programming standards and conventions. It does not alter the source code; rather, it advises the user when the standards and conventions have been violated. According to [RADC 1976b], there are four standards applied: (1) documentation standards, (2) format standards, (3) design standards limiting module size and restricting the use of inefficient instructions, and (4) structural standards of top-down design.

The tool is being used on the Site Defense program and is now in evaluation phase.

The tool is written in FORTRAN and runs on the Honeywell 600/6000 series under GCOS and within the GCOS environment that is simulated under RADC's MULTICS. It requires 40K words utilizing chain overlay. It could with "medium difficulty" [Mark 1976] be transported to a different machine and operating environment. A users manual and a maintenance manual will be published in the near future.

RXVP
Developed, maintained, and owned by General Research Corp.

RXVP is a program path execution analyzer for use on FORTRAN programs. It is currently being modified for JOVIAL J3B-2. It produces a post-execution report that includes instruction frequency counts, path execution frequencies, decision paths, and label, procedure, and module trace paths. It may be used as a program debugger, and it includes a cross-referencer. It may be used in batch mode or interactively.

RXVP has been available for a few years, but has not been used yet by the Air Force. It will be used at the Defense Mapping Agency within a year [LaMonica 1976]. An evaluation of RXVP has been made by the Air Force Avionics Lab (Report No. AFAL-TR-75-242).

RXVP is currently implemented on the CDC 6000 and 7000 (Scope 3.4) and the IBM 370 (MVS), and is being implemented on the Univac 1108 and Honeywell 6180. The supplier quotes a price of $5000 for conversion to any machine [LaMonica 1976]. All language-dependent features have been isolated in the front end [Wishart 1976]. RXVP served as a model for JAVS, but none of the actual code was used. RXVP is coded in structured FORTRAN (IFTRAN) and will be converted to STRUCTRAN-1 prior to Air Force delivery.

A users guide and a reference guide to RXVP are available from General Research Corp.

JAVS
(Jovial Automated Verification System)
Developed by General Research Corporation for RADC

JAVS is a program-path-execution analyzer that operates on successfully-compiled JOCIT JOVIAL (J3) source modules and produces a variety of reports to aid in testing those modules. It operates in batch mode only, and in a series of separately-submitted steps it provides the following facilities: (1) It produces various kinds of source program reformattings and analyzes the source to create tables for use in subsequent steps (Step 1). (2) It identifies the source program's pathways between each of its conditional branches, ("Decision-to-Decision", or "D-D" pathways), creates a version of the source program instrumented for tracing of flow and variable assignments, and produces detailed reports and graph-like lists indicating the degree to which various parts of the program have been tested (Steps 2, 3, 4). (3) It prepares reports describing the dynamic relations among the various modules (including library modules) of a multiple-module system (Step 5). (4) From the JAVS library prepared by previous steps, and an execution of the subject program, it prepares additional reports counting a variety of post-mortem statistics on the degree of path execution, (for one run or cumulative for all runs) and various other execution-time summaries (Step 6).

JAVS is written in and for JOCIT JOVIAL (J3) running on the Honeywell 6180 under GCOS. It contains approximately 33K lines of J3 source code excluding comments. JAVS requires the use of JOCIT version of J3 rather than the standard version of J3 documented in [AFM 100-24 1967], because the JAVS source code trace instrumentation utilizes JOCIT's MONITOR statement which is absent in standard J3. Modifying JAVS so that it would process another language (or another dialect of JOVIAL), would entail designing another method of obtaining trace information and redoing the front end. The extent of that effort could not be determined, even for non-JOCIT J3. One rough estimate that was obtained indicated that a modification to handle J3-B would entail 2-3 months work for two people. A modification for J73 could presumably utilize J73's !TRACE facility instead of JOCIT's MONITOR statement, but the extent of other J3 dependencies is unknown. JAVS also has dependencies on the Honeywell OS interface, but the extent of these is also unknown.

JAVS has been applied to itself, but it has not yet been used outside of RADC. It is currently in a six-month test effort that is scheduled to conclude on November 10, 1976. Some minor enhancements are being added to JAVS as the tests proceed. SAC plans to use JAVS for future maintenance of its FMIS programs. Boeing has inquired about using JAVS for AWACS and the B-1 programs [Lombardo 1976].

A User's Guide [Brooks et. al. 1975] and Reference Manual [Brooks and Gannon 1975] have been published by General Research Corporation.

SIMON
(Software Implementation Monitor)
Developed by MITRE for AF

SIMON is a system intended to serve two purposes: (1) to provide technical and managerial visibility of the software development process, and (2) to provide systematic and consistent collection of data for research into factors affecting software quality and cost. SIMON includes for information collection a prepostprocessor for JOVIAL (J3) source programs. In addition, information is supplied to the system directly through user prepared project sheets. Five classes of reports are made available by SIMON. These are: 1) System Structure, 2) Estimates vs. Actuals (both time and cost), 3) Project Schedules, 4) Project Status Reports, and 5) Error and Discrepancy Summaries.

A prototype of the system is operational on Honeywell Series 600/6000 computers running under GCOS. SIMON requires a maximum of 50K words of storage on those systems. It was released for use at RADC in early 1976. As yet, it has only been used in conjunction with one small development project. "An Experience-Based Critique of the SIMON Prototype", available from RADC, constitutes an evaluation of its value in that effort. The project is felt to have been too small to reflect the true value of the tool, however [Fleischer 1976].

SIMON is written in JOVIAL (J3) and COBOL using IDS. It is felt that transport to a new machine would be fairly straightforward as long as it supported these languages and some facility similar to IDS. The support of new languages could be accomplished by rewriting the pre- and post-compilers (approximately 30% of the system).

The system is similar to IBM's Program Support Library (PSL). Its uniqueness rests in the facilities it provides for error classification and computation of a complexity measure for each module.

The specifications for SIMON have been published by the MITRE Corporation [Corrigan et al. 1975]. Captain Samuel Ruple is responsible for the system supplied to RADC.

EFFIGY

Developed and Owned by IBM

EFFIGY is an experimental symbolic executor meant for use in testing and debugging. It permits entire classes of input to be tested simultaneously by representing variables by symbolic, as opposed to numerical values. The system supports several standard debugging commands permitting the user to set breakpoints, save his environment, explicitly assign values to variables, and trace the execution. Additional commands allow the selection of program branches, assertion of variable properties, and attempted proof of these assertions.

EFFIGY supports a small subset of PL/I. The expansion of this subset is being attempted but requires a great deal of work. Support of other languages would similarly involve a large effort.

The system runs on an IBM/370 168 under VM/370 and uses CMS. Its portability is greatly limited due to the use of CMS for file accessing and editing.

James C. King has been responsible for the development of EFFIGY. The work is continuing at the Thomas J. Watson Research Center in New York. Two papers have appeared detailing the concepts of the system [King 1975 and 1976].

Other Tools

The tools listed below are ones for which more detailed information either was unavailable or could not be collected due to the limited scope of this study.

COBOL Structured Programming Pre-Compiler

This tool permits programmers to write programs in a version of ANS COBOL X3.23-1968 that is "augmented" with various constructs supporting structured programming. The output of the tool is that program re-written in standard COBOL. It is an RADC product.

AED

AED ia a high-order language system that contains various compiler-writing tools. It was developed at the Massachusetts Institute of Technology and is now distributed by Softech, Inc.

XPL

XPL is a compiler-writing system consisting of an implementation language and various programs that aid in the construction of compilers. It was developed by W. M. McKeeman of the University of California at Santa Cruz and by J. J. Horning and D. B. Wurtman of the University of Toronto.

CSS II and GPSS V

CSS II and GPSS V are simulation languages developed by IBM for the 360/370 systems [Pomeroy 1972].

PTT (Program Testing and Translation)

PTT is an execution path analyzer for FORTRAN programs. It is a McDonnell-Douglas Aerospace Corp. product.

Consistency Checker

This tool checks a design data base for consistency with a requirements data base. Both data bases must be in URL/URA. The tool is being built by TRW for RADC and is not yet completed.

PPL (Programming Production Library)

PPL is a library system developed by IBM. It was used as a basis for the design of the PSL for WWMCCS, but lacks various automatic facilities found in PSL.

PDL (Program Design Language)

PDL is a non-compilable pseudo-code created by McDonnell-Douglas for purpose of informally stating program logic in a way that is easily expandable by hand into conventional source programs.

NLS

NLS is a sophisticated text editor system that contains a variety of modularized capabilities as subsystems, including text reformatting, word indexing, graphics display, and numerous techniques for input and output of data. It was developed by the Stanford Research Institute and is available on the ARPANET.

TECO (Text Editor and COrrector)

TECO is a programmable text editor created by Bolt Beranek and Newman, Inc., and available on the ARPANET [Feinler 1976].

FORTRAN Pre-processors

A list of over fifty extant FORTRAN pre-processors has been published in [Reifer 1976].

SFORTRAN

SFORTRAN is a FORTRAN preprocessor used at SAMTEC.

DISSECT

This tool is an experimental symbolic executor for FORTRAN programs. DISSECT was developed at the University of San Diego and is supervised and funded by McDonnell-Douglas.

DECA

This tool is an on-line aid for specifying top-level software design in tree form and for checking state transitions at the various levels in the tree. DECA was designed at Boeing Computer Services, Inc., and is in use there.

RPE

This tool is an automated program prover being developed by SRI for RADC. It uses Floyd's inductive assertion method and processes a limited subset of JOCIT JOVIAL. It is described in RADC Report No. RADC-TR-76-58.

GYPSY

This tool is a program design language developed and implemented at the University of Texas (Austin). It is being enhanced to provide verification and validation facilities.

AMPIC

This tool is an aid that restructures, flowcharts, and symbolically executes a source program. It is developed by Logicon, Inc., and written in a SNOBOL dialect for the IBM 360/370.

FLOW and AUTOFLOW

FLOW and AUTOFLOW are FORTRAN flowcharters available on the ARPANET. AUTOFLOW has an option for assembly language source and for producing an output tape for a plotter [Feinler 1975].

COBOL Debug

This tool is an interactive debugger for COBOL programs. It is an IBM product, and it operates under IBM's Time Sharing Option (TSO) [Pomeroy 1972].

BAIL

BAIL is an interactive runtime debugger written at Stanford University and available on the ARPANET [Feinler 1975].

SIMDDT, COBDDT, and DDT

These tools are program debuggers for SIMULA 67, COBOL, and FORTRAN respectively. All are for the PDP-10, and DDT is available on the ARPANET and in a version for the PDP-11.

QUALIFIER

QUALIFIER is a FORTRAN program path execution analyzer developed by Computer Software Analysis, Inc., for the CDC 6600 [Kennedy 1974].

SOFTOOL

SOFTOOL is a FORTRAN program path execution analyzer designed by General Research Corp. for the CDC 6400 [Kennedy 1974].

PCEP and ZYGO

PCEP and ZYGO are FORTRAN program path execution analyzers developed by General Electric [Kennedy 1974].

APADS and FORDAP

APADS and FORDAP are FORTRAN program path execution analyzers developed at the University of California at Los Angeles for the IBM 360 [Kennedy 1974].

PROGTIME, PROGFORT, and FETE

PROGTIME, PROGFORT, and FETE are FORTRAN program path execution analyzers developed at Stanford University for the IBM 360 [Kennedy 1974].

PET (Program Evaluator and Tester) and SPY

PET and SPY are FORTRAN program path execution analyzers developed by McDonnell-Douglas [Kennedy 1974].

PACE (Produce Assurance Confidence Evaluator), NODAL, and TDEM

PACE, NODAL, and TDEM are FORTRAN program path execution analyzers developed by TRW Systems Group, Inc., for CDC, Univac, and IBM systems [Kennedy 1974].

CCVS (COBOL Compiler Validation System)

This tool is a COBOL compiler validator developed by the Navy for the National Bureau of Standards.

NBS FORTRAN Test Programs

This tool is a compiler validator developed by the National Bureau of Standards.

DES (Design and Evaluation System)

This tool is a program design language and simulation modeling system developed by Honeywell and implemented on MULTICS [Graham et. al. 1973]. The system is not fully operational.

4.0 SOFTARE DEVELOPMENT, HOL STANDARDIZATION, AND TOOLS

4.1 Tools and the Computer Program Life Cycle

Numerous minor differences exist among the various definitions of what the exact stages are in the software life cycle. The development of a large system is a complex process in which several activities occur at different times and at different rates, depending on the nature of that particular portion of the system. At least one fact is clear: a large software system can have a "lifetime" of fifteen years or longer, and decisions made in the early stages of development affect what remains in operation many years later.

Air Force Regulation 800-14 (Vol. II) identifies two kinds of life cycles of systems involving computers. There is the life cycle of the total system (System Acquisition Life Cycle) and the Computer Program Life Cycle, which occurs during one or more phases of the System Acquisition Life Cycle.

System Acquisition Life Cycle	Computer Program Life Cycle
1. conceptual phase	1. analysis phase
2. validation phase	2. design phase
3. full-scale development phase	3. coding and checkout phase
4. production phase	4. test and integration phase
	5. installation phase
5. deployment phase	6. operation and support phase

The Computer Program Life Cycle "may span more than one system acquisition life cycle phase, or occur in any one phase . . . ." It "will occur at least once for each CPCI [computer program configuration item] during the system acquisition life cycle" [AFR 800-14 1975]. In the development of a large system, the early stages of one component computer program's development may coincide with later stages in the development of another program in a different part of the system. One task that software tools can assist in is the assurance that the various parts of the system will not fail to integrate for some unpredicted reason.

Of all the categories and sub-categories of tools in the taxonomy in Section Two, only the Application Program Design aids are applied in solely one or two phases in the Computer Program Life Cycle (i.e., in the analysis phase and the design phase). Most of the tools have a broad range of potential applicability -- not just in coding and checkout, but also in integration, installation, and the maintenance portion of the operation and support phase. Code analyzers, for example, are used to check out not only original coding, but also any changes to programs occurring throughout operation and support. Program testers such as the environmental simulators are used not only for original checkout and test phases, but also for assuring that all maintenance changes will have the desired affect. Similar reasons give the translators and the related translator development and language maintenance aids a broad range of applicability not only over their own life cycles but also over the life cycles of the programs which those translators process.

For these reasons, plus the fact that the computer program life cycles themselves interact in sometimes intricate ways, it would be foolish and wasteful to design tools for use only by the person or agency involved in a limited portion of a program life cycle. There is a tendency now to build tools for program development that can also be used (without modification) for maintenance (e.g., SDVS). The same general requirements exist, and if the tool can be made sufficiently versatile, a significant saving should result from using one set of tools for both. Even within one project, a given tool may be used not only by several different persons, but even by several different agencies or contractors. The developer of a software system is often different from the maintainer of the system, and both are different from the users.

For each tool, it will take experience in using the tool before the necessary versatility in design can be achieved. SDVS, for example, was consciously designed for use in maintenance as well as the coding and checkout phases. The file-handling mechanism of SDVS deliberately made it difficult or impossible to delete old files. (During maintenance, particularly, one might wish to return to a previously-useful file.) But SDVS users in the initial coding and checkout phases have a need to create temporary, short-term files, knowing in advance that they will soon wish to delete them. This added versatility is one of the planned

modifications to SDVS, and it illustrates the role that actual experience can have in helping to reveal a need for modification in a reusable tool. Many Air Force tools have yet to be used in a variety of operational phases, so this kind of tool modification will in all likelihood have to be made to more than one tool before its full potential for versatility and reusability can be achieved.

Because the application program aids used in program coding and checkout can also be reused in the later life cycle phases (given the appropriate versatility in design and ease of use), the middle and later phases of the life cycle are much better supported by tools than the early phases of analysis and design. However, the cost of correcting software errors is least in the preliminary design phase. Correcting the error in the coding and debugging phase can be over twice as expensive. The relative cost of error-correction increases with each successive life cycle phase, and if the error goes uncorrected until the operation phase, correcting it can be between 10 and 88 times as expensive to correct as it would have been in the preliminary design phase [Press 1976]. It is ironic and unfortunate that the earliest phases of the life cycle are the ones least well-supported by tools.

There are several reasons for this lack of tools applicable to the analysis and design phases. Most obvious, of course, is the fact that tools process code and if the system is not yet designed, there is no code to process. The idea of creating a "preliminary" code (in the form of a simulation model or a specification in a program design language to which tools may be applied) is a comparatively new one, and the most effective means of doing this is still under study. Also, it is only in the last few years that statistics have become available indicating how costly undetected analysis and design errors can be. Hopefully, as more emphasis is placed on the early design stages, better techniques and tools will be developed.

It is clear that more tool-development efforts are needed in this area. The Air Force Program Specification No. CP 07877-96100A [RADC 1976a] stipulates that a yet-to-be created <u>program design language</u> is to be used instead of flow charts for program design. It is reported in [La Padula and Loring 1976] that in at least one Range Support application, the preliminary top-level design is coded and tested in FORTRAN, then after the design is approved, the FORTRAN code is discarded and the actual software is coded in assembly language. It remains to be seen, however, exactly what the optimum analysis and design techniques are, and the extent to which tools can contribute to those early phases of the software cycle.

4.2 Tools and HOL Standardization

Commonality among software tools, including computers and related equipment is necessary in order to obtain the full benefits of HOL standardization. Because tools are inherently more independent of specific applications than are the application programs to which they are applied, it is in the area of software tools where a significant portion (perhaps the greatest portion [Fisher 1976]) of cost savings can occur. According to [Callender et. al. 1976], the amount of Air Force software cost that goes for tools (now about 20%) could be reduced to between 5% and 10% by standardizing on the software development environment and using a common HOL.

Standardizing on common HOLs makes some tools more necessary than others. According to [DiNitto 1976], the following tools are needed in order to adequately develop, maintain, and control a high-order language:

(1) A compiler-writing tool aids in the production of numerous compilers that process the standard language. The availability of a compiler is already the most important way in which tools affect the choice of a language for Air Force projects over the broad spectrum of applications [LaPadula and Loring 1976]. If compilers for a common, standard HOL cannot be made available readily, that HOL will not achieve wide use. A compiler-writing tool is also helpful in supporting other tools that parse the language. JOCIT and CWS are Air Force tools in this category. Compiler-writing tools should not be placed in the hands of users of the language; rather, they should be available only to the agency that controls the language.

(2) A compiler validator can help ensure that each compiler exactly implements the intended language.

(3) A standard language specification helps ensure that no language issues remain to be resolved by persons implementing compilers. The SEMANOL language specification language and language specification processor are designed to fulfill this task, but their use is not yet practical [Dreisbach et. al. 1976].

(4) A statistics collector aids in maintaining a file documenting the extent to which the various language features are used, thereby enabling the controller of the language to maintain it more effectively. A statistics collector for JOVIAL J3 is currently under development by RADC.

One group or agency should have responsibility for control and administration of these and related tools. A detailed proposal for a "language control facility", and a detailed examination of its related tools, is the subject of the study performed by Computer Sciences Corporation [CSC 1976].

The choice of what languages are to be standards also affects what tools are needed for applications software development. Any modern language supporting structured programming reduces the need for pre-processors and flow-charters. Good modern languages incorporate facilities such as module interface checking and compool control, which would otherwise have to be handled outside the language by separate tools.

In addition to providing direct cost savings, tool commonality would give greater visibility to the role of tools in the software development process. Use of unknown or unpublicized tools by individual contractors prevents the wider community from profiting from experiences with new tool concepts and retards an understanding of how tools can best aid in the software life cycle processes. Air Force procurement practices should be modified, where necessary, to assure delivery to the Air Force of all tools used in the creation and validation of computer programs.

Finally, tools are themselves software products, and they have their own life cycle. HOL commonality permits greater use of tools on tools. Both SDVS and the DAIS application software are written in J73/I, and a valuable by-product of SDVS was that the SDVS developers helped debug the J73/I compiler in the process of SDVS development. Also, JCVS has been used on the J73/I compiler as well as on J3 compilers. Like other application programs, tools need regular maintenance under proper administrative controls.

4.3 Trends in Tool Design

Modern tools are being designed in the form of "packages" of tools that are supplied together and incorporate several tool capabilities. Some of these combine tool capabilities from several categories in the taxonomy. The Program Support Library for WWMCCS, unlike its predecessor libraries includes pre-processor, source-program reformatting, and test case generation aids. SDVS provides several kinds of simulation facilities as well as diagnostic and file-manipulation capabilities.

Modern compilers provide numerous "tool" features including diagnostics and static analysis aids such as counters, reformatters, and cross-referencers. Incorporating these tools into the compiler offers the advantage that once the compiler is standardized and generally available, so are the tools. The compiler produced by JOCIT has numerous standard diagnostics in addition to uniform optimizations and a facility for turning off compilation of "debugging" portions of source code [CSC 1975].

This combination of tool capabilities within a single package offers several advantages. It is more efficient to use one tool instead of a set of independent tools. For tools requiring parsing of the source code, a tool package has to make only one parse. In avionics applications, using an integrated tool package such as SDVS enables users to run the same simulation models in several phases of development.

Achieving well-integrated packages is not an easily-achievable goal, however. If two integrated packages are used in the same system, some duplication may result. The current version of JAVS, for example, duplicates many of the syntactic analysis activities of the JOCIT compiler. At present, JAVS can be used only with JOCIT. Whether most JAVS facilities should be part of the compiler or part of a separable verification system is a moot point.

In one sense, combining tools into integrated packages makes the tools less easily portable. A package is most conveniently designed for a particular kind of application, and the component tools are not easily "unpluggable" from the package for use separately. In SDVS, for example, one could conceivably build a different statement level simulator (to convert SDVS to handle a different source language) or build a different interpretive computer simulator (to convert SDVS to simulate a different machine). But one cannot remove the statement-level simulator or the interpretive computer simulator for use independently. In addition, incorporating a reusable tool in an integrated tool package can make that package less portable. SDVS, rather than using a custom-designed text editor, uses the text editor of the DEC-10 machine on which SDVS is designed for use at DAIS. The fact that SDVS does not have its own text editor makes it harder to transport SDVS to a system (e.g., IBM) with a different text editor. Ironically, designing a system around a reusable component can detract from the reusability of the total system.

These difficulties indicate that tool development and refinement, like the process of standardization itself, is a lengthy process. The benefits of present and past efforts are just now being seen, and they reveal that much more work lies ahead.

GLOSSARY OF ACRONYMS

ACM	Association for Computing Machinery
AFAL	Air Force Avionics Laboratory
AFM	Air Force Manual
AFR	Air Force Regulation
ANSI	American National Standards Institute
CSC	Computer Sciences Corporation
CTCC	(Joint Services) Command Control Technology Center
DAIS	Digital Avionics Information System
DEC	Degital Equipment Corporation
DMA	Defense Mapping Agency
ESD	Electronic Systems Division
GRC	General Research Corporation
GMAP	The Assembly language and Macro facility for the HIS 600/6000 series computers
HIS	Honeywell Information Systems, Inc.
HOL	High-Order Language
IBM	International Business Machines, Inc.
JOCIT	JOVIAL Compiler Implementation Tool
MITRE	The MITRE Corporation
NBS	National Bureau of Standards
RADC	Rome Air Development Center
SAMSO	Space and Missile Systems Organization
SAMTEC	Space and Missile Test Center
SOVS	Software Design and Verification System
TRW	TRW Systems, Inc.
WWMCCS	World Wide Military Command and Control System

REFERENCES

[AFM 100-24 1967]

Standard Computer Programming Language for Air Force Command and Control Systems, Air Force Manual 100-24, 1967.

[AFR 800-14 1975]

Acquisition and Support Procedures for Computer Resources in Systems, Air Force Regulation 800-14 (Vol. II), 1975.

[Brooks and Gannon 1975]

Brooks, N. B., and C. Gannon, JAVS Computer Program Documentation: Reference Manual, General Research Corp., Report No. CR-3-465 (Vol. II), 1975.

[Brooks et. al. 1975]

Brooks, N. B., C. Gannon, and R. J. Urban, JAVS Computer Program Documentation: User's Guide, General Research Corp., Report No. CR-3-465 (Vol. I), 1975.

[Callender et. al. 1976]

Callender, E. D., M. Feliciano, and L. D. Jennings, SAMSO Computer Language and Software Development Environment Requirements, Aerospace Corporation, Report No. SAMSO-TR-290, 1975.

[Carlson 1976]

Carlson, William, NBS, personal telephone communication, September, 1976.

[Chalstrom 1976]

Chalstrom, Judy, C. S. Draper Laboratory, personal communication, October, 1976.

[Cheng 1976]

Cheng, L., Software Engineering Techniques and Tools: Language and Life Cycle Considerations, MITRE Corporation, 1976.

[Cohen and Zuckerman 1974]

Cohen, J., and C. Zuckerman, "Two Languages for Estimating Program Efficiency", Communications of the ACM, Vol. 17, No. 6 (June, 1974), 301-308.

[Corn 1974]

Corn, B. C., "A Compiler Development System", Proceedings of the Aeronautical Systems Software Workshop, Dayton, Ohio, April 1974, pp. 418-424.

[Corrigan et. al. 1975]

Corrigan, A. E., R. J. Fleischer, R. W. Spitler, and J. E. Sullivan, Specifications for SIMON, a Software Implementation Monitor, MITRE Corporation, Report No. MTR-3056, July, 1975.

[CSC 1975]

JOCIT Compiler Users Manual, Computer Sciences Corporation (under contract F30602-72-C-0467 with RADC), Revision 3, September, 1975.

[CSC 1976]

Language Control Facility Study, Computer Sciences Corporation, June, 1976.

[DiNitto 1976]

DiNitto, Samuel A., RADC, and Richard Motto, RADC, personal telephone communication, October, 1976.

[DiNitto and Motto 1976]

DiNitto, Samuel A., RADC, and Richard Motto, RADC, personal telephone communication, October, 1976.

[Draper 1976a]

PALEFAC Functional Design Specification, C. S. Draper Laboratory, Inc., January, 1976.

[Draper 1976b]

PALEFAC Detailed Design Specification, C. S. Draper Laboratory, Inc., CSDL-DAIS-4, April, 1976.

[Dreisbach et. al. 1976]

Dreisbach, T. A., J. L. Felty, I. Greenberg, R. Hartman, A. Kramer, C. Mikkelsen, M. Roth, and L. Weissman, High-order Language Technology Evaluation Final Report, Intermetrics, Inc., Report No. IR-203-2, October, 1976.

[Dunbar 1975]

Dunbar, T. L., JOCIT JOVIAL Compiler Implementation Tool, Computer Sciences Corporation, Final Report, RADC-TR-74-322, January, 1975.

[ESD 1975a]

URL User's Manual, Air Force Electronic Systems Division, Report No. ESD-TR-75-88, 1975.

[ESD 1975b]

URA User's Manual, Air Force Electronic Systems Division, Report No. ESD-TR-75-355, 1975.

[Feinler 1975]

Feinler, Elizabeth J., ed., ARPANET Resource Handbook, Network Information Center, NIC 23200, Stanford Research Institute, September, 1975.

[Fisher 1976]

Fisher, D. A., A Common Programming Language for the Department of Defense -- Background and Technical Requirements, Institute for Defense Analyses, Paper P-1191, June, 1976.

[Fleischer 1976]

Fleischer, R. J., MITRE Corp., personal telephone communication, October, 1976.

[Graham et. al. 1973]

Graham, R. M., G. R. Clancy, Jr., and D. B. DeVaney, "A Software Design and Evaluation System", Communications of the ACM, Vol. 16, No. 2 (February, 1973), 110-116.

[Hollowich and Borasz 1976]

Hollowich, M., and F. Borasz, The Software Design and Verification System (SDVS) -- An Integrated Set of Software Development and Management Tools, NAECON '76 Record, 1976, pp. 920-926.

[IBM 1976]

WWMCCS Programming Support Library (PSL) Phase I Users Manual, IBM Corp., October, 1976.

[Kennedy 1974]

Kennedy, James E., A Survey of Automated Computer Program Verification Tools, Aerospace Corporation, Report No. TOR-0075 (5112)-1, August, 1974.

[King 1975]

King, J. C., "A New Approach to Program Testing", Proceedings of the 1975 International Conference on Reliable Software, pp. 228-233.

[King 1976]

King, J. C., "Symbolic Execution and Program Testing", Communications of the ACM, Vol. 19, No. 7 (July, 1976), 385-395.

[LaMonica 1976]

LaMonica, Frank, RADC, personal telephone communication, October, 1976.

[LaPadula and Loring 1976]

LaPadula, L. J., and P. L. Loring, Air Force Programming Languages: Standards, Use, and Selection, MITRE Corporation, ESD-TR-76-140, August, 1976.

[Lesak 1976]

Lesak, Linda, Aerospace Corp., personal telephone communication, October, 1976.

[Lombardo 1976]

Lombardo, Larry, RADC, personal communication, September, 1976.

[Luppino and Smith 1974]

Luppino, F. M., and R. Smith, Structured Programming Series, Vol. V, Programming Support Library (PSL) Functional Requirements, IBM Corporation, Final Technical Report, AD R003-339, RADC-TR-74-300, July, 1974.

[Luppino and Tinanoff 1974]

Luppino, F. M., and N. Tinanoff, Structured Programming Series, Vol. VI, Programming Support Library (PSL) Program Specifications, IBM Corporation, Final Technical Report, AD A007-796, RADC-TR-74-300, November, 1974.

[Mark 1976]

Mark, Donald, RADC, personal communication, October, 1976.

[Melanson 1976]

Melanson, Ann, ESD, personal telephone communication, October, 1976.

[Pomeroy 1972]

Pomeroy, J.W., "A Guide to Programming Tools and Techniques," IBM Systems Journal, Vol. 11, No. 3, (1972), 234-254.

[Press 1976]

Press, B., "Automated Tools: A Philosophy and Approach", TRW Systems presentation at DoD V&V Conference, August, 1976.

[Price 1976]

Price, Ray, TRW Systems, Inc., personal telephone communication, September, 1976.

[RADC 1976a]

A Program Design Language Description, RADC Computer Software Development Specification No. CP 07877-96100A, Appendix 40, April, 1976.

[RADC 1976b]

Technology Products, Rome Air Development Center Information Sciences Division, RADC/ISFA, September 1976.

[Ramamoorthy 1965]

Ramamoorthy, C., "Discrete Markov Analysis of Computer Programs," ACM 20th National Conference, August, 1965, pp. 386-391.

[Reifer 1975a]

Reifer, D. J., "Automated Aids for Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, pp. 131-142.

[Reifer 1975b]

Reifer, D. J., Interim Report on the Aids Inventory Project, Aerospace Corporation, Report No. SAMSO-TR-75-184, July, 1975.

[Reifer 1976]

Reifer, D. J., "The Structured FORTRAN Dilemma", SIGPLAN Notices, Vol. 11, No. 2 (February, 1976), 30-32.

[Ruple 1976]

Ruple, Capt. Samuel L., RADC, personal telephone conversation, September, 1976.

[Stover 1976]

Stover, Robert E., RADC, personal telephone communication, October, 1976.

[Trainor 1976]

Trainor, W. Lynn, AFAL, personal communication, October, 1976.

[TRW 1973a]

SEMANOL Specification of JOVIAL, TRW Systems Group, Inc., Redondo Beach, Calif., March, 1973.

[TRW 1973b]

A Standard for Language Implementation, SEMANOL Reference Manual, TRW Systems Group, Inc., Redondo Beach, Calif., 1973.

[TRW 1975]

SEMANOL(73) Specification of JOVIAL(J73), Final Technical Report, RADC-TR-75-211, Vol. III (of four), TRW Systems Group/Rome Air Development Center, Air Force Systems Command, Griffiss AFB, New York, 1975.

[TRW 1976]

Software Design and Verification System (SDVS) Users' Manual, Phase Three, TRW Systems Group, Air Force Avionics Laboratory DAIS Software Group, June, 1976.

[White 1976a]

White, Douglas, A BASIC Statistics Collector, RADC Report No. RADC-TR-76-9, March, 1976.

[White, 1976b]

White, Douglas, RADC, personal telephone communication, October, 1976.

[Wishart 1976]

Wishart, Richard, GRC, personal telephone communication, October, 1976.